CMPS 312

Navigation

Dr. Abdelkarim Erradi

CSE@QU

Navigation

The act of moving between screens of an app to complete tasks

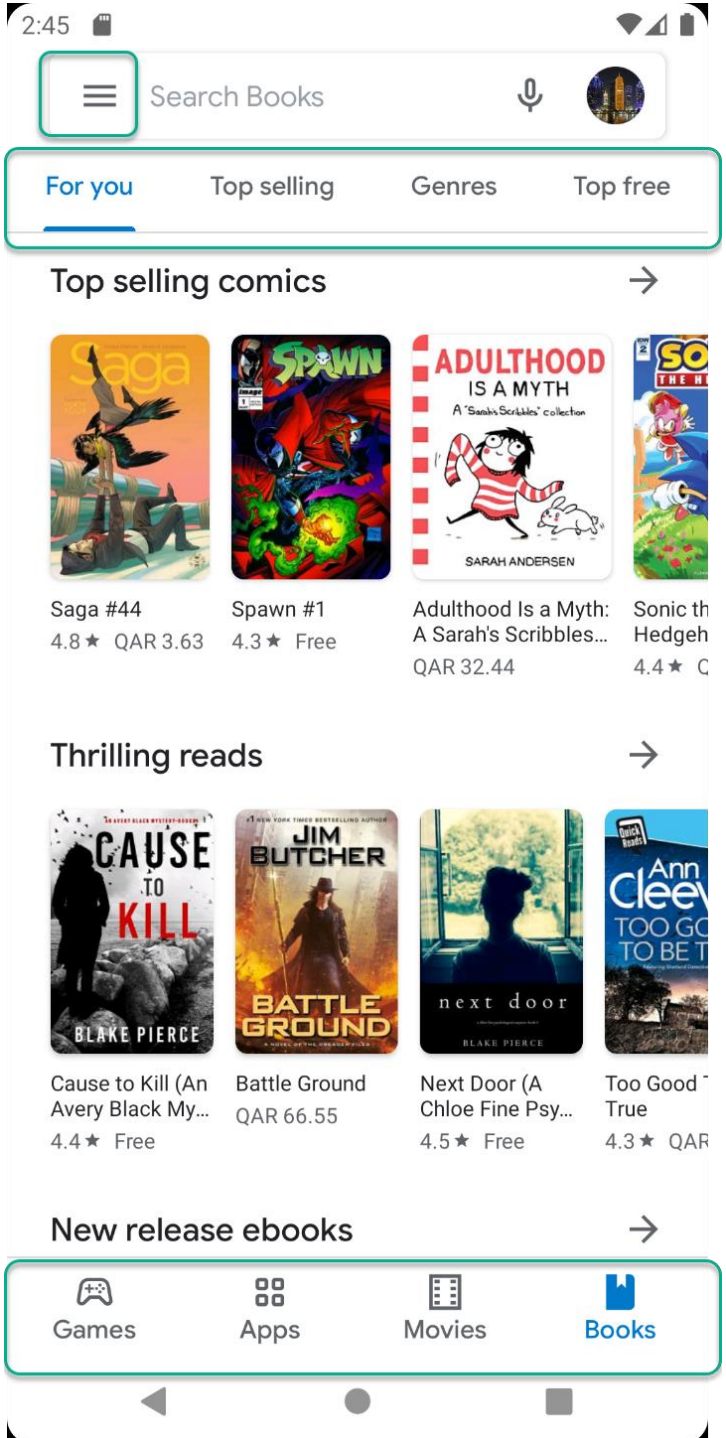
Designing effective navigation = Simplify the user journey

Outline

1. Navigation UI
2. Jetpack Compose Navigation
3. Alert Dialog

Navigation UI:

Components: App Bars, Floating Action Button, Navigation Drawer

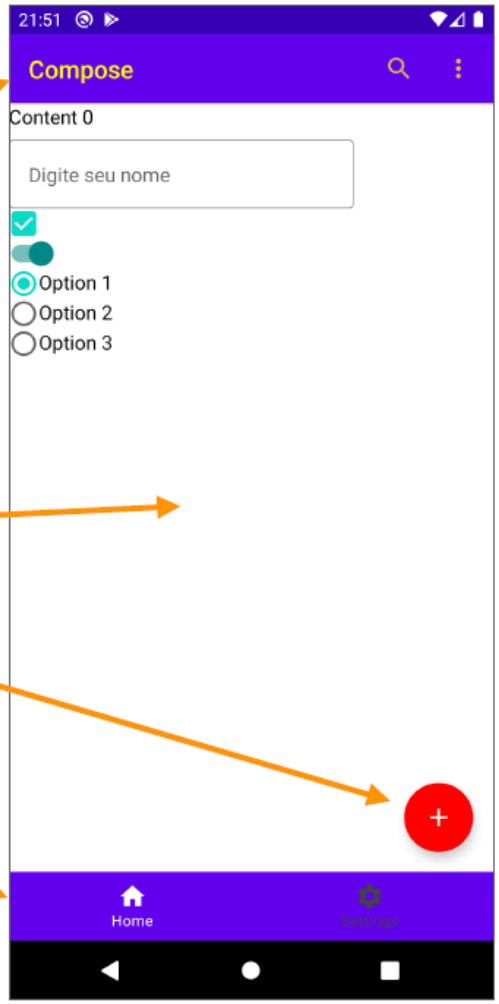


Back

Scaffold

- **Scaffold** is a **Slot-based** layout
- Scaffold is template to build the entire screen by adding different UI Navigation components (e.g., topBar, bottomBar, floatingActionButton)

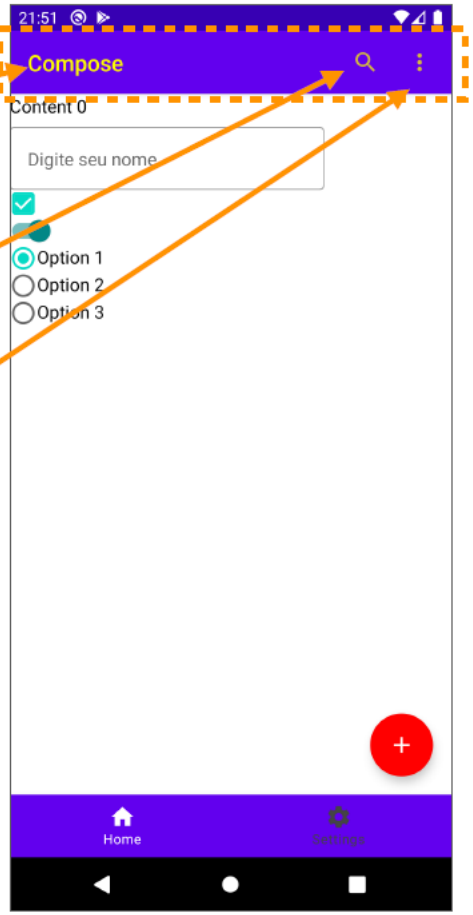
```
Scaffold(
    topBar = {...},
    floatingActionButton = {...},
    bottomBar = {...}
) {...}
```



TopAppBar

- Info and actions **related to the current screen**
- Typically has Title, Menu items, Drawer button / Back button

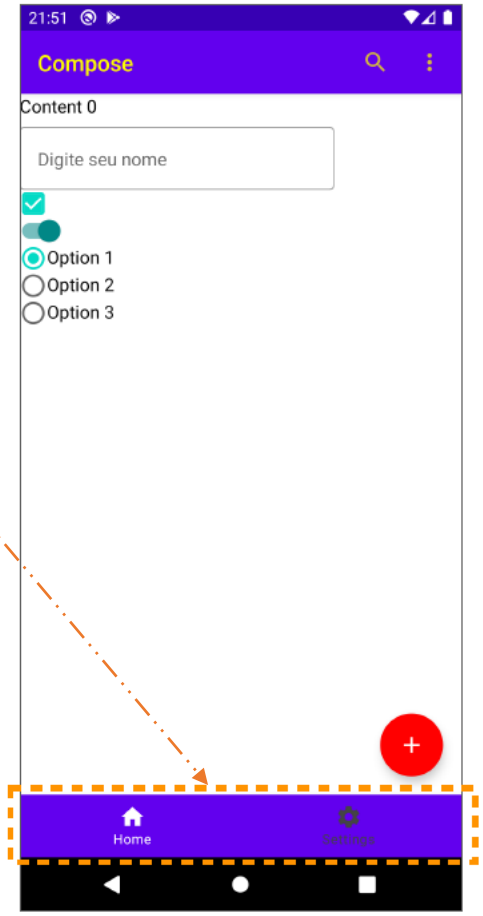
```
TopAppBar(
    title = { Text(text = "Compose") },
    backgroundColor = MaterialTheme.colors.primary,
    contentColor = Color.Yellow,
    actions = {
        IconButton(onClick = {}) {
            Icon(Icons.Default.Search, "Search")
        }
        IconButton(
            onClick = { … }
        ) {
            Icon(Icons.Filled.MoreVert, "More")
            DropdownMenu(…)
        }
    }
)
```

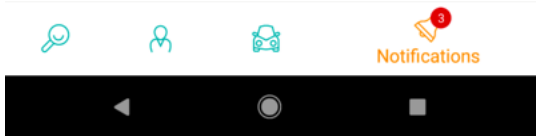


BottomAppBar

- Allow movement between the app's primary **top-level destinations** (3 to 5 options)
- Each destination is represented by an icon and an optional text label. May have notification badges

```
BottomAppBar(
    backgroundColor = MaterialTheme.colors.primary,
    content = {
        BottomNavigationItem(
            icon = { Icon(Icons.Filled.Home) },
            selected = selectedTab == 0,
            onClick = { selectedTab = 0 },
            selectedContentColor = Color.White,
            unselectedContentColor = Color.DarkGray,
            label = { Text(text = "Home") }
        )
        BottomNavigationItem(…)
    }
)
```

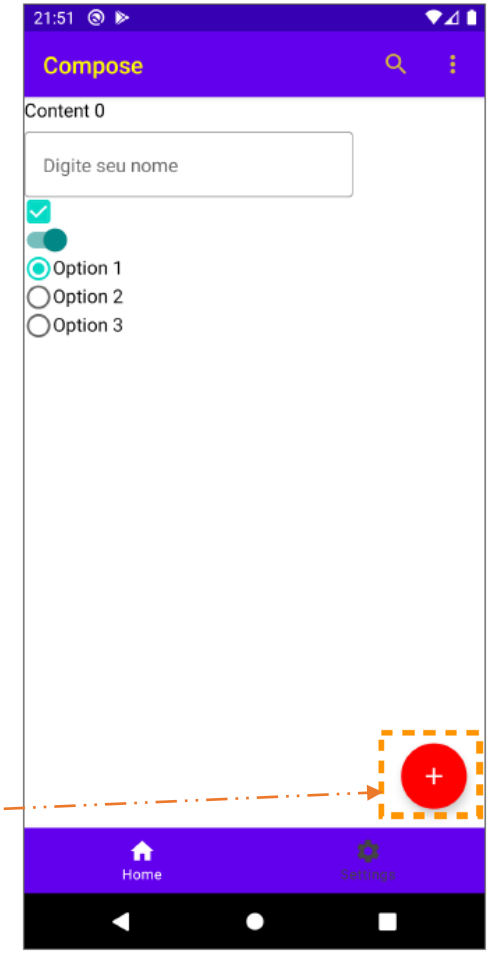




Floating Action Button (FAB)

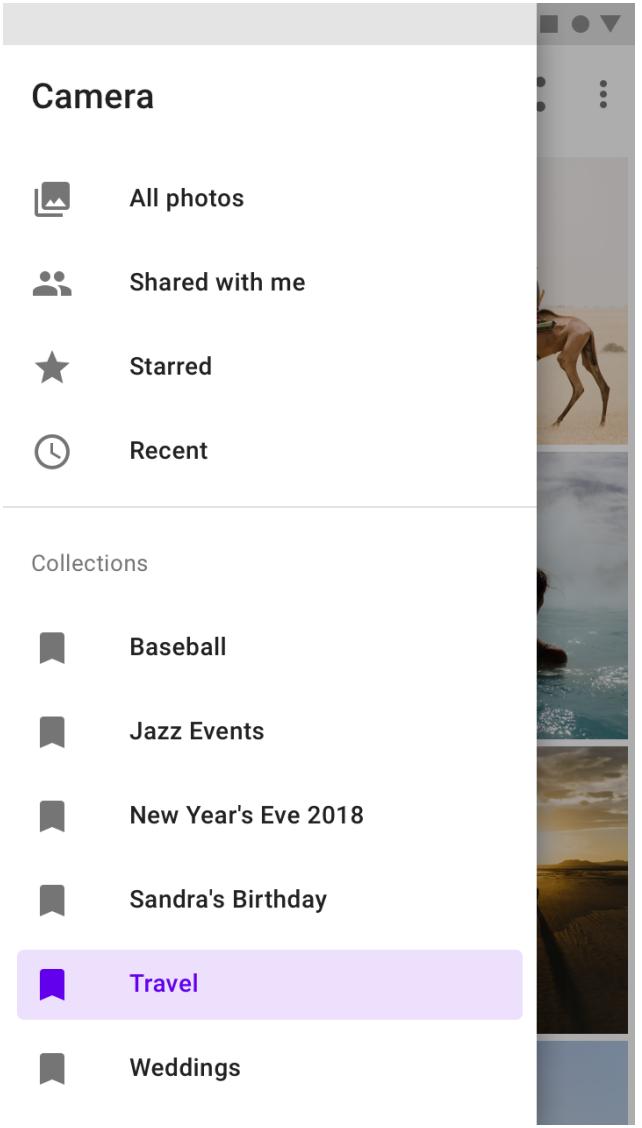
- A FAB performs the primary, or most common, action on a screen, such as drafting a new email
  - It appears in front of all screen content, typically as a circular shape with an icon in its center.
  - FAB is typically placed at the bottom right

```
FloatingActionButton(
    onClick = { … },
    backgroundColor = Color.Red,
    contentColor = Color.White
) {
    Icon(Icons.Filled.Add, "Add")
}
```



Navigation Drawer

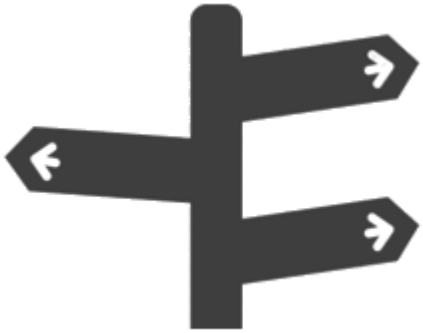
- Navigation Drawer provides access to **primary destinations**, such as switching accounts
  - Recommended for five or more top-level destinations
  - Quick navigation between unrelated destinations
- The drawer appears when the user touches the drawer icon ☰ in the app bar or when the user swipes a finger from the left edge of the screen



Snackbar

Jetpack Compose Navigation

A framework for navigating between destinations within an app



Back

https://developer.android.com/jetpack/compose/navigation

Single Activity with Multi-Screens

- App UI = { 1 Activity + Multi-Screens }
  - A Screen is a composable that represents **a portion of the UI**
- The Navigation Component enables implementing Single Activity App with the ability to navigate between the app screens (i.e., composables)
- Requires the following dependency in app module's `build.gradle` file:

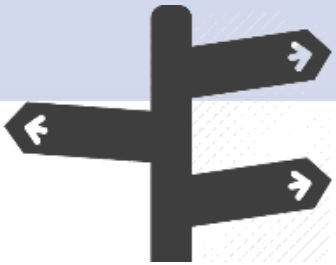
```
implementation "androidx.navigation:navigation-compose:2.4.0-alpha09"
```

- ff

Key Components

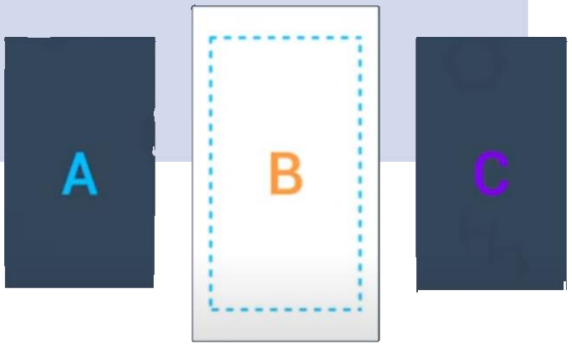
Navigation Graph

- Defines the app navigation **routes** (how users can move between screens of the app): **possible routes** a user can take through the app



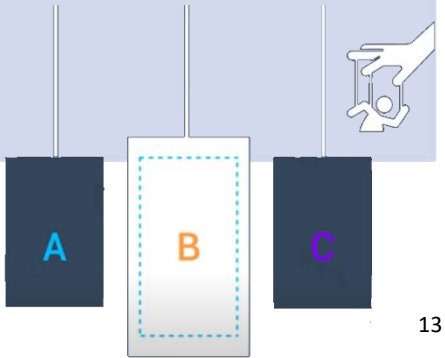
NavHost

- A container where screens will be displayed as the user navigate through the app



NavController

- Manages the transition and **loading** of destination screens into the NavHost as the user navigates through the app
- Keeps track of the **back stack** of visited screens



Creating a NavHost

- Add **`NavHost`** to the Main Activity to define a **navigation graph** that specifies the possible **routes** within the app + the startDestination route
  - Route is a String that defines the **path** that leads to a specific destination (i.e., a screen implemented as composable). Each destination should have a unique route
- The Nav Graph is defined using the **`composable`**() fun to define each **route** and the associated **screen**

```
NavHost(navController = navController, startDestination = "profile") {
    composable("profile") { Profile(/*...*/) }
    composable("friendslist") { FriendsList(/*...*/) }
    /*...*/
}
```

Navigate to a destination using NavController

- The **`NavController`** is created (typically in the Main Screen) using the **`rememberNavController`**()

```
val navController = rememberNavController()
```

- Then use the **`navigate(`*`destinationRoute`*`)`** method to navigate to a specific destination
  - The requested destination screen will be loaded in the **`NavHost`**

```
@Composable
fun Profile(navController: NavController) {
    /*...*/
    Button(onClick = { navController.navigate("friends") }) {
        Text(text = "Navigate next")
    }
}
```

Navigate with arguments

- To pass arguments to a destination e.g., get profile for user 123 `navController.navigate("profile/1234")`
  - First add the argument placeholder to the destination route
  - The user profile destination takes a userId argument to determine which user to display

```
NavHost( ...) {
    composable("profile/{userId}") {...}
}
```

- By default, all arguments are parsed as strings. You can specify another type by using the arguments parameter

```
composable("profile/{userId}",
    arguments = listOf(navArgument("userId") { type = NavType.IntType })
) { backStackEntry ->
    Profile(navController, backStackEntry.arguments?.getInt("userId"))
} // Extract the NavArguments from the NavBackStackEntry
```

Adding optional arguments

- Optional arguments must be explicitly added to the `composable()` as a query parameter **`?argName={argName}`**
  - They must have a defaultValue set, or have nullability = true (which implicitly sets the default value to null)

```
composable("profile?userId={userId}",
    arguments = listOf(navArgument("userId") { defaultValue = "me" })
) { backStackEntry ->
    Profile(navController, backStackEntry.arguments?.getString("userId"))
}
```

NavOptions - popUpTo and popUpTo inclusive

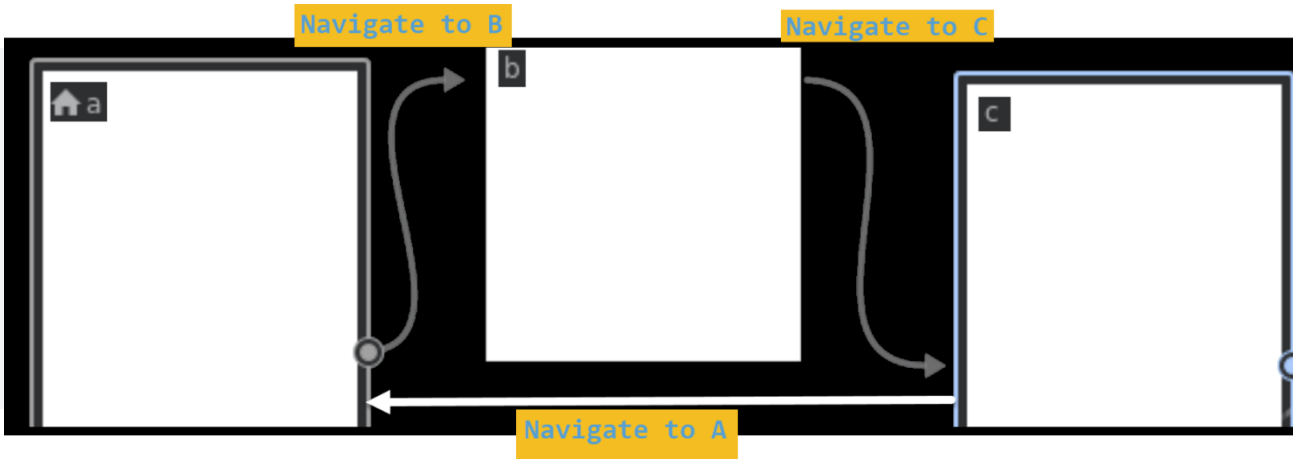
- By default, **`navigate()`** adds the new destination to the back stack. To modify this behavior, pass **navigation options** to **`navigate()`** call
  - **`popUpTo(route)`** pop off previously visited destinations from the back stack (up to the specified route)
- For example, after a login flow, you should **pop off all the login-related destinations** of the back stack so that the Back button doesn't take users back into the login flow
  - It should go back to the Home Screen while removing all visited destinations from the back stack
  - If `inclusive = true` the destination specified in popUpTo should also be removed from the back stack

Navigation Options: popUpTo & launchSingleTop

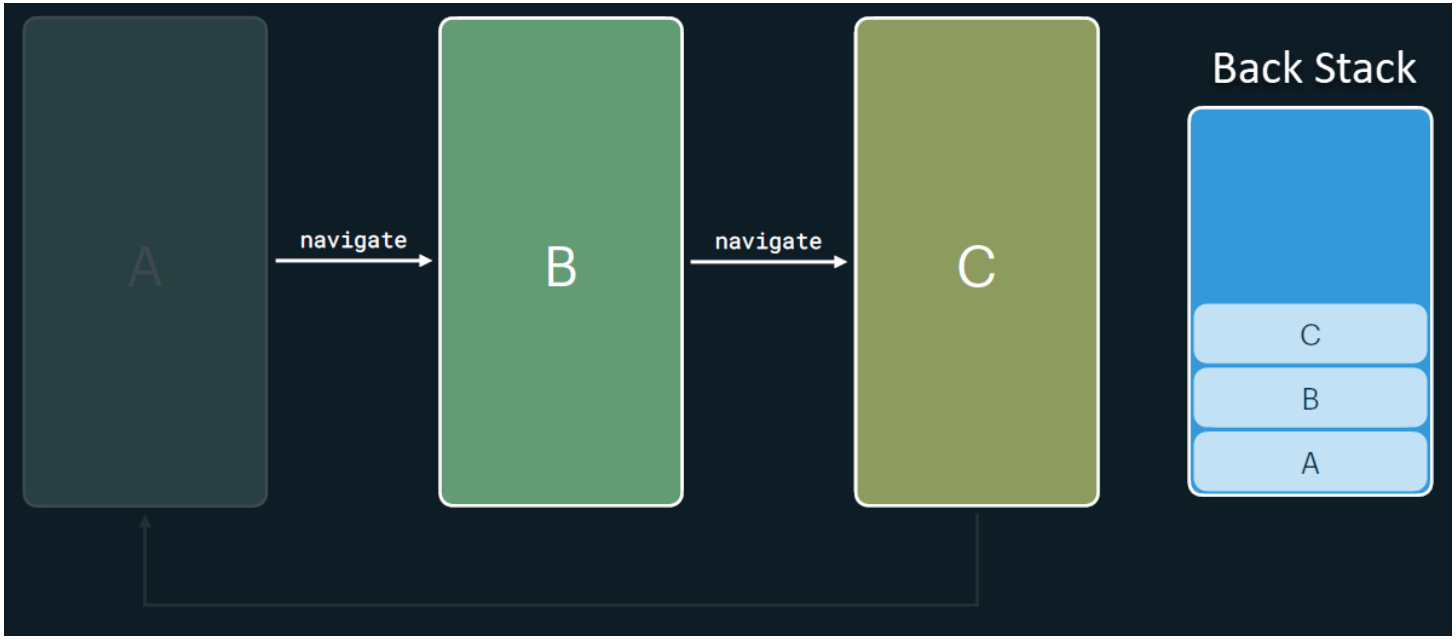
```
// Pop everything up to the "home" destination off the back stack before
// navigating to the "friends" destination
navController.navigate("friends") {
    popUpTo("home")
}

// Pop everything up to and including the "home" destination off
// the back stack before navigating to the "friends" destination
navController.navigate("friends") {
    popUpTo("home") { inclusive = true }
}

// Navigate to the "search” destination only if we’re not already on
// the "search" destination, avoiding multiple copies on the top of the
// back stack
navController.navigate("search") {
    launchSingleTop = true
}
```

popUpTo Example

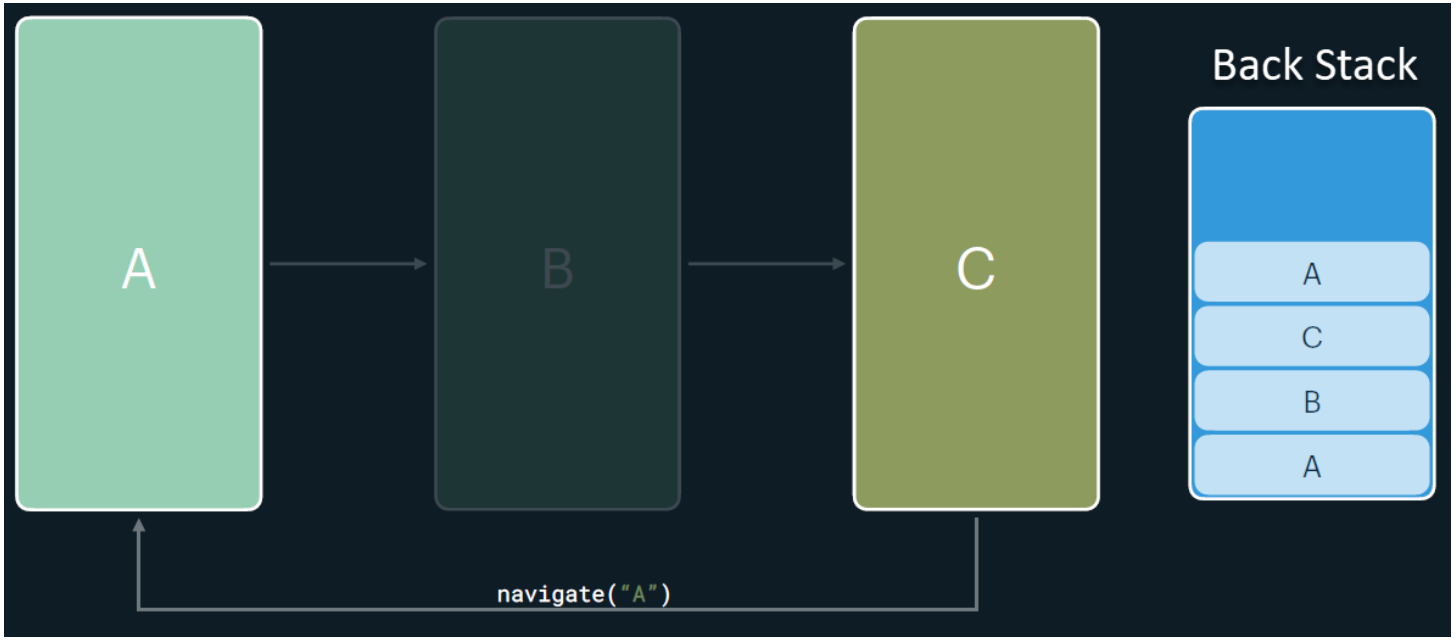
```
navController.navigate("A") {
    popUpTo("A") {
        inclusive = true
    }
}
```

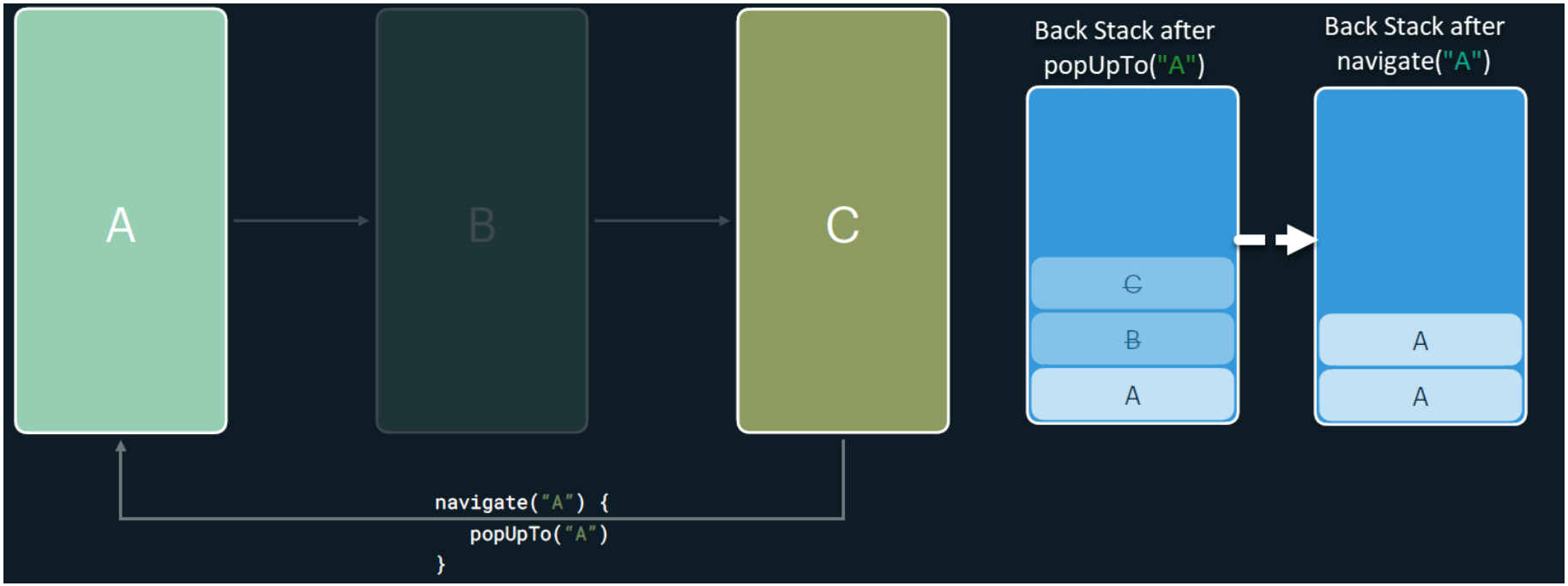


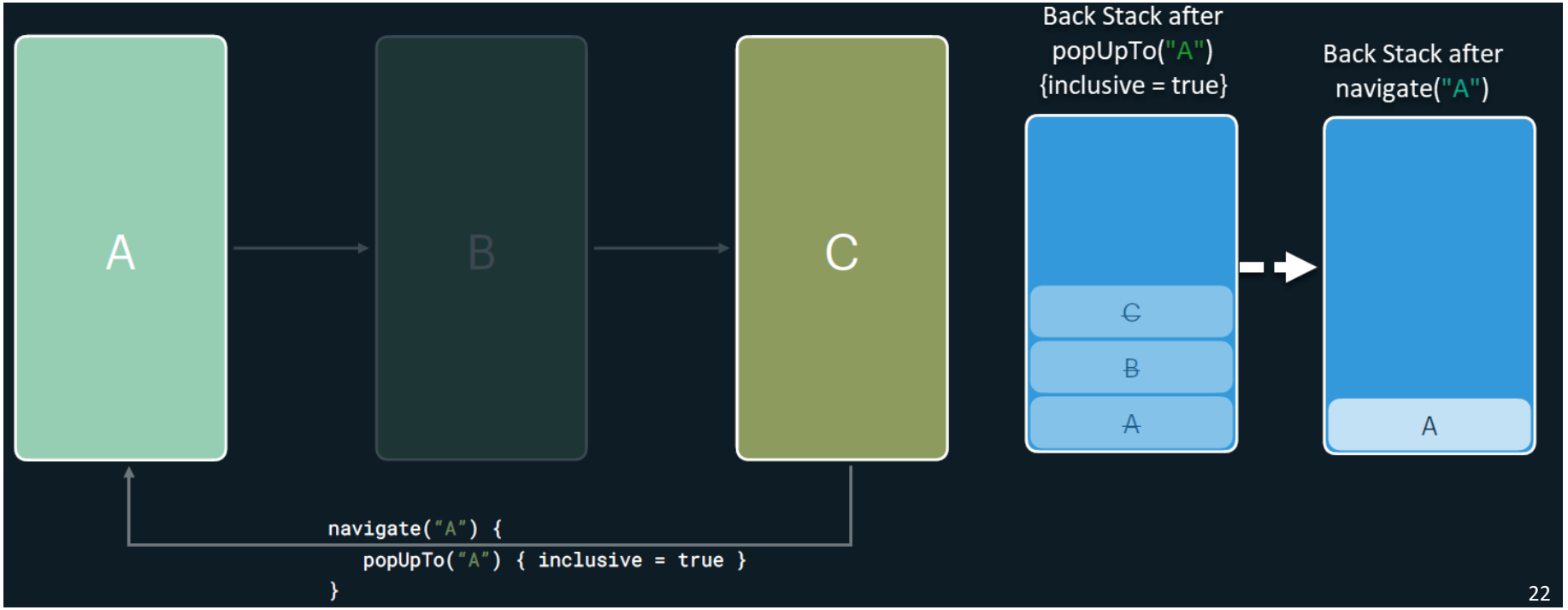
- After reaching C, the back stack contains (A, B, C). When navigating back to A, we also **popUpTo A**, which means that we remove B and C from the stack as part of the call to **navigate**(**"A"**)
  - With inclusive= true, we also pop off that first A of the stack to avoid having two instances of A




```
navController.navigate("A")
```




A
B
C
Back Stack after popUpTo("A")
C
B
A
Back Stack after navigate("A")
A
A
navigate("A") {
popUpTo("A")
}

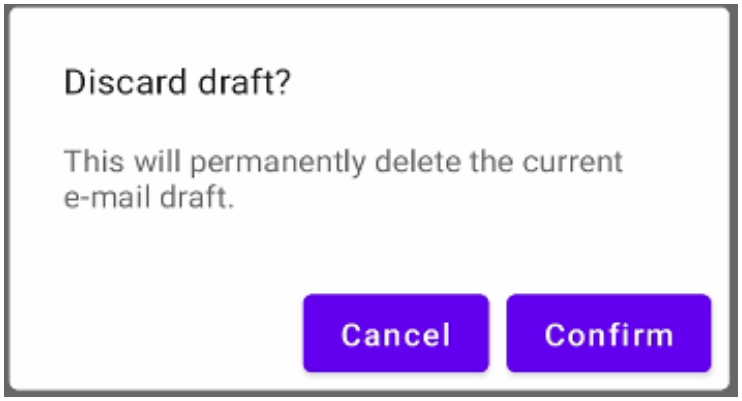

A
B
C
Back Stack after popUpTo("A") {inclusive = true}
C
B
A
Back Stack after navigate("A")
A
navigate("A") {
popUpTo("A") { inclusive = true }
}

Alert Dialog

Alert Dialog

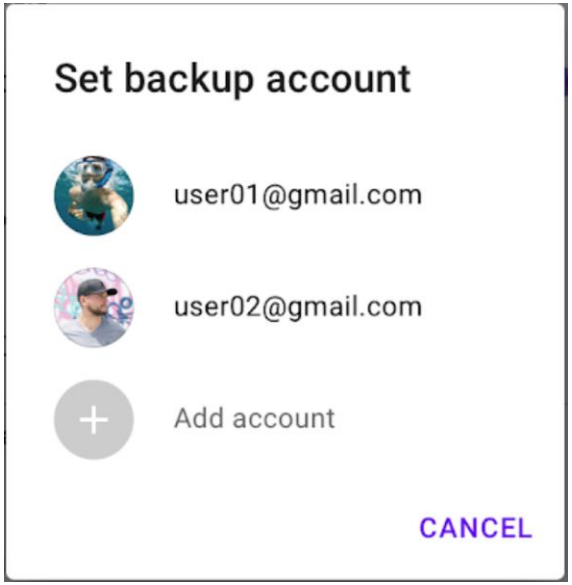
- Alert dialog is a Dialog which interrupts the user with urgent information, details or actions
- Dialogs are displayed in front of app content
  - Inform users about a task that may contain **critical information** and/or **require a decision**
  - Interrupt the current flow and remain on screen until dismissed or action taken. Hence, they should be used sparingly
- 3 Common Usage:
  - **Alert dialog**: request user action/confirmation. Has a title, optional supporting text and action buttons
  - **Simple dialog:** Used to present the user with a list of actions that, when tapped, take immediate effect.
  - **Confirmation dialog:** Used to present a list of single- or multi-select choices to a user. Action buttons serve to confirm the choice(s)

Alert Dialog

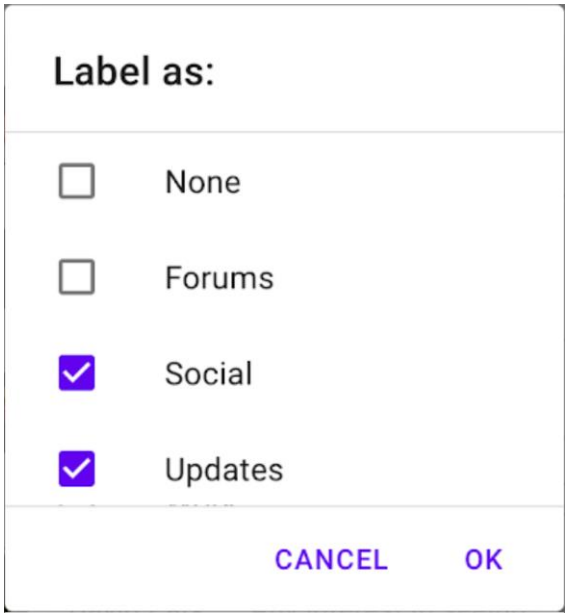



```
AlertDialog(
        onDismissRequest = {
           // Dismiss the dialog when the user clicks outside the dialog
           // or on the back button
             onDialogOpenChange(false)
        },
        title = { Text(text = title) },
        text = { Text(text = message) },
        confirmButton = {
            Button(
                onClick = {  onDialogResult(true) }) {
                Text(text = "Confirm")
            }
        },
        dismissButton = {
            Button(
                onClick = {  onDialogResult(true) }) {
                Text("Cancel")
        }
    }
  )
}
```

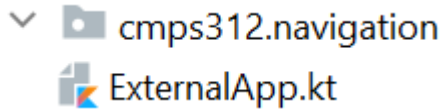
Simple dialog



Confirmation dialog (multi choice)



Routing to External App



- **Intent** can be used to route a request to another app
  - Specify an **Action** and the **Parameters** expected by the action
  - Implicit intents can be handled by **a component in the system** registered to handle that intent type

- **Dial a number:**

```
val intent = Intent(Intent.ACTION_DIAL).apply {
    data = Uri.parse("tel:$phoneNumber")
} context.startActivity(intent)
```

- **Open a Uri**

```
val intent = Intent(Intent.ACTION_VIEW,
Uri.parse("https://www.qu.edu.qa"))
startActivity(intent)
```

- **Share content**

```
val intent = Intent(Intent.ACTION_SEND).apply {
    putExtra(Intent.EXTRA_TEXT, content)
    type = "text/plain"
}
context.startActivity(Intent.createChooser(intent, "Share via"))
```

- Other common intents discussed here

Resources

- Jetpack Compose Navigation
  - https://developer.android.com/jetpack/compose/navigation

- Jetpack Compose Navigation codelab
  - https://developer.android.com/codelabs/jetpack-compose-navigation